20 | 生成订单信息之二：业务逻辑复杂，怎么做性能优化?

2021-05-05 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 28:05 大小 25.72M

你好，我是高楼。

在上节课中，我们针对生成订单信息这个接口做了三个阶段的分析定位和优化动作，让 TPS 变得正常了一些。不过，系统资源并没有完全用起来，这个接口显然还有优化的空间。因为高老师说很过多遍，**在性能优化的过程中，我们要把资源都用起来。**

关于“把资源用起来”这一理论，我希望你能明白的是，**我们在性能环境中做优化，把资源用起来是为了看系统的最大容量在哪里。这并不意味着，你可以在生产环境中让硬件使用到这种程度。**

对于一个不可控的系统容量来说，资源使用率高，极有可能导致各种问题出现。所以，安全稳妥起见，很多生产环境的资源利用率都是非常低的，倘若用得超过了 20%，运维都得

半夜惊出一身冷汗。

而我们在性能环境中的测试结果，要想给生产环境配置一个比较明确并且可以借鉴的结论，就必须先去分析生产的业务容量，然后再来确定当生产业务容量达到峰值的时候，相应的硬件资源用到多少比较合理。

不过，在我们的优化环境中，我们可以通过把一个系统用起来，来判断软件的容量能力。所以，我们接着上节课的内容，再进入到第四阶段。你将看到在业务逻辑复杂的情况下，我们该怎么做优化。

闲言少叙，直接开整。

第四阶段

在解决了前面三个不正经的问题之后，我们现在可以正常分析时间消耗到哪去了，只要解决了快慢的问题，我们才能进而解决资源没有用起来的问题。所以，我们先来拆分响应时间，同样，我们也不做全局监控分析，因为..... 哥嫌累。

拆分响应时间

之前很多次我们都在用 APM 来拆分响应时间，感觉没什么新意，这次我用日志来拆分一下时间。

Gateway：

复制代码

```
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate
10.100.79.93 - - [04/Feb/2021:00:13:17 +0800] "POST /mall-order/order/generate

```

Order：

复制代码

```
10.100.79.106 - - [04/Feb/2021:00:13:31 +0800] "POST /order/generateOrder HTTP
10.100.79.106 - - [04/Feb/2021:00:13:31 +0800] "POST /order/generateOrder HTTP
10.100.79.106 - - [04/Feb/2021:00:13:31 +0800] "POST /order/generateOrder HTTP
10.100.79.106 - - [04/Feb/2021:00:13:31 +0800] "POST /order/generateOrder HTTP
10.100.79.106 - - [04/Feb/2021:00:13:31 +0800] "POST /order/generateOrder HTTP
```

我们先不用看后面的服务，因为从这个接口往后就直接到数据库了，我们先来看一下应用本身有没有问题。

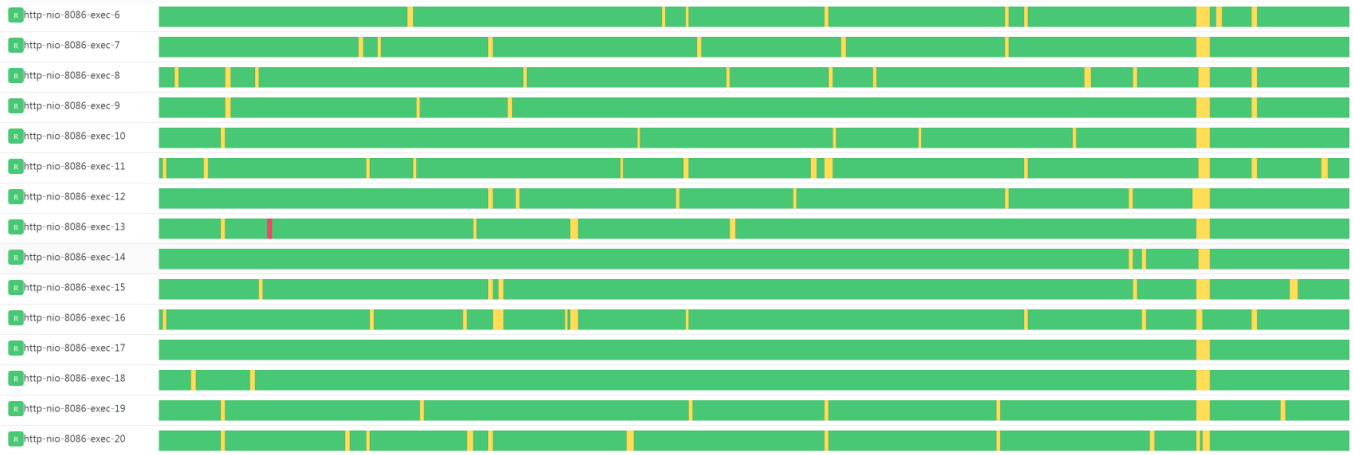
为了让你看得清楚一点，这里我只截取了部分数据，但并不是说我们只看这些就够了。在项目中的话，你可以通过写脚本或其他的方式自己做响应时间的统计。

从上面的信息可以看到，这个接口的整个响应时间是 150ms 左右，而在 order 服务上就消耗了 90 毫秒。所以，下面我们要分析：为什么在 order 上会消耗这么久的时间。

定向监控分析

要想知道 Order 服务的时间消耗，那显然，我们得知道 Order 应用中的线程都在做什么动作，所以我们先直接来分析 Order 的栈。

通过 Spring Boot Admin，我们可以查看到线程的整体状态：



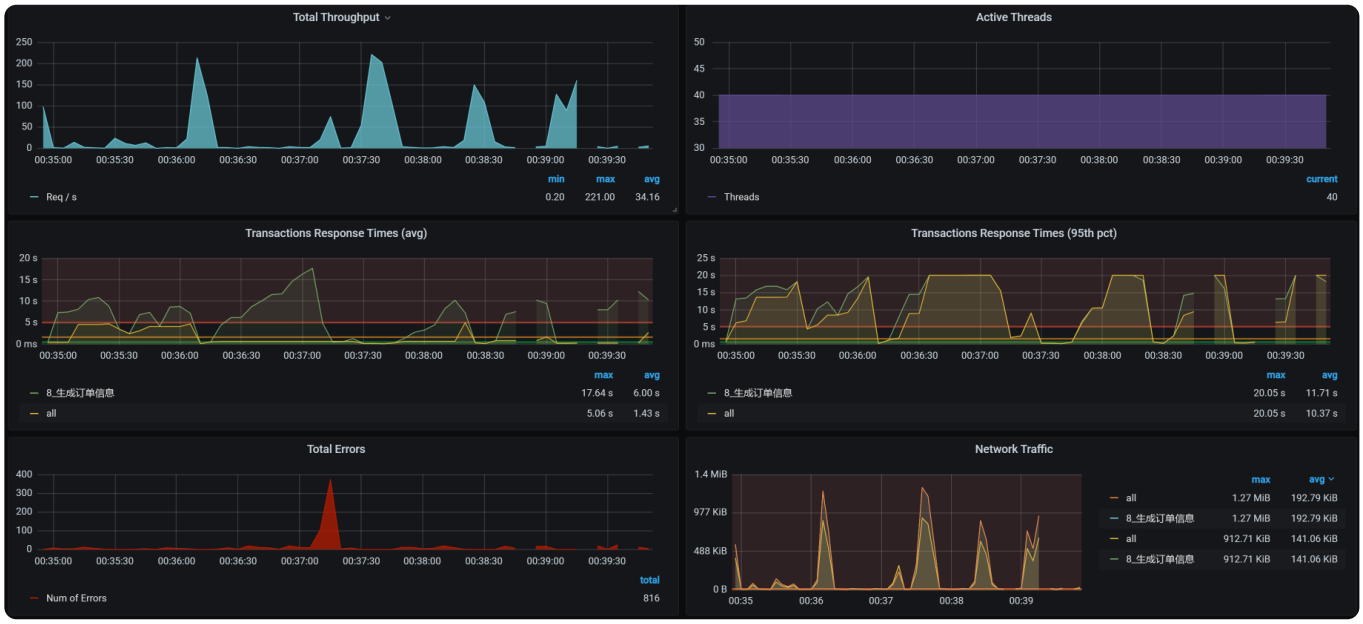
你看，线程确实比较繁忙。至于这些线程在做什么，我们通过栈的内容可以知道，然后再进一步确定优化的方向。

但是，由于系统资源还没有用到上限，我们得先调整一下 Tomcat 的线程数，把它加大一些，争取让 Order 应用把硬件资源用起来。

复制代码

```
原值:
max: 20
修改为:
max: 100
```

我们看一下调整后的结果：



没想到，性能更差了……这乱七八糟的曲线和想像中的优美曲线完全不一致呀！

事实证明，偷懒是绕不过去坑的，我们只有再次查看响应时间消耗到了哪里。

于是，通过用各个服务的日志拆分响应时间，我发现在 Member 服务上有这样的日志（为了让你看清楚点，我截取了一些时间消耗比较大的日志，请注意一下哦，这是我们改了 Order 的 Tomcat 线程池之后的结果）：

复制代码

```
10.100.69.248 - - [04/Feb/2021:00:37:15 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:17 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:17 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:15 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:17 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:20 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:20 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:20 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:21 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:22 +0800] "GET /sso/feign/info HTTP/1.1"
```

```
10.100.69.248 - - [04/Feb/2021:00:37:22 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:22 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:22 +0800] "GET /sso/feign/info HTTP/1.1"
10.100.69.248 - - [04/Feb/2021:00:37:23 +0800] "GET /sso/feign/info HTTP/1.1"
```

显然，这个 Member 服务的响应时间太长了。而在生成订单信息这个接口中，也确实调用了 Member 服务，因为要使用 Token 嘛。既然是 Order 的 Tomcat 线程池加大了，导致 Member 服务响应如此之慢，那我们就有理由作出判断：Order 之所以消耗时间长，是因为 Member 服务不能提供 Order 请求时的快速响应。通俗点讲，就是 Member 的性能差。

要想分析 Member 性能为什么差，我们其实可以直接到 Member 上打印栈信息来看看，这是高老师有时候偷懒的做法。

而我们一直在讲，完整的分析逻辑应该是先看全局监控数据，再看定向监控数据。所以，高老师在这里，勤快一点。我们通过全局监控数据来看看整体的资源消耗：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	4.01 week	7.64 GiB	4	0.42	8.69%	23.17%	29.95%	0 B/s	109.95 kB/s	285	59	626.74 kbps	442.85 kbps
k8s-master-2	k8s-master-2	6.59 week	7.64 GiB	4	1.05	20.54%	26.03%	29.95%	0 B/s	94.59 kB/s	317	42	1.23 Mbps	1.23 Mbps
k8s-master-3	k8s-master-3	6.60 week	7.64 GiB	4	0.20	7.39%	24.41%	29.95%	0 B/s	95.81 kB/s	295	117	299.69 kbps	446.48 kbps
k8s-worker-1	k8s-worker-1	1.01 week	15.51 GiB	6	0.42	7.77%	50.83%	68.65%	208.70 kB/s	450.10 kB/s	28	81	4.16 Mbps	5.80 Mbps
k8s-worker-2	k8s-worker-2	2.05 day	15.51 GiB	6	0.03	3.04%	14.86%	30.26%	23.02 kB/s	5.63 kB/s	28	147	59.67 kbps	46.41 kbps
k8s-worker-3	k8s-worker-3	14.29 hour	15.51 GiB	6	1.84	12.94%	21.90%	49.54%	0 B/s	283.90 kB/s	29	284	8.19 Mbps	8.26 Mbps
k8s-worker-4	k8s-worker-4	5.51 week	15.51 GiB	6	2.42	15.02%	37.82%	47.51%	9.21 kB/s	43.95 kB/s	52	4 K	1.94 Mbps	3.46 Mbps
k8s-worker-5	k8s-worker-5	5.50 week	15.51 GiB	6	1.13	17.09%	34.44%	37.01%	13.81 kB/s	27.27 kB/s	29	148	1.64 Mbps	1.85 Mbps
k8s-worker-6	k8s-worker-6	5.50 week	15.51 GiB	6	0.50	15.33%	21.34%	39.61%	23.02 kB/s	11.75 kB/s	30	106	827.07 kbps	792.50 kbps
k8s-worker-7	k8s-worker-7	5.20 week	15.51 GiB	4	1.74	17.90%	29.29%	42.16%	23.02 kB/s	52.84 kB/s	28	214	2.73 Mbps	2.64 Mbps
k8s-worker-8	k8s-worker-8	4.03 week	15.51 GiB	4	60.38	97.12%	50.24%	59.13%	0 B/s	194.62 kB/s	31	259	11.16 Mbps	14.28 Mbps
k8s-worker-9	k8s-worker-9	5.20 week	15.51 GiB	6	1.16	13.41%	26.73%	29.95%	4.60 kB/s	5.54 kB/s	28	73	11.14 Mbps	24.04 Mbps

worker-8 的 CPU 资源居然用到了这么高！这说明我们在前面增加 Order 的 Tomcat 线程数是有价值的。现在，瓶颈点到了另一个地方，也就是我们的 Member 服务。

既然 worker-8 的资源使用率高，那我们就来看看它上面有什么 Pod，不难看出 Member 就在 worker8 上：

复制代码

```
[root@k8s-master-2 ~]# kubectl get pods -o wide | grep k8s-worker-8
elasticsearch-client-0                  1/1     Running     0          38h
monitor-mall-monitor-d8bb58fcb-kfbcj    1/1     Running     0          23d
skywalking-oap-855f96b777-5nxll         1/1     Running     6          37h
skywalking-oap-855f96b777-6b7jd         1/1     Running     5          37h
svc-mall-admin-75ff7dcc9b-8gtr5         1/1     Running     0          17d
svc-mall-demo-5584dbdc96-fskg9          1/1     Running     0          17d
```

```
svc-mall-member-5fc984b57c-bk2fd            1/1     Running     0          12d
[root@k8s-master-2 ~]#
```

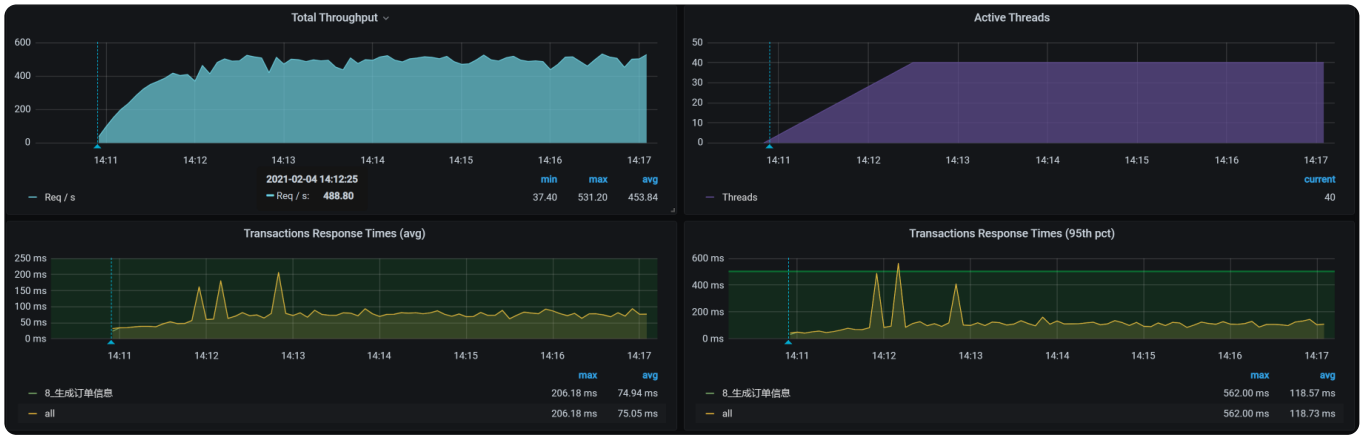
同时，我们还能发现，这个节点上有不少服务，而这些服务都是比较吃 CPU 的，并且在压力过程中，还出现了 sy cpu 消耗很高的情况，我截两个瞬间的数据给你看看，一个是 sy cpu 高的情况，一个是 us cpu 高的情况，具体如下所示：

复制代码

```
- sys cpu高的情况
[root@k8s-worker-8 ~]# top
top - 00:38:51 up 28 days,  4:27,  3 users,  load average: 78.07, 62.23, 39.14
Tasks: 275 total,  17 running, 257 sleeping,   1 stopped,   0 zombie
%Cpu0  :  4.2 us, 95.4 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.4 s
%Cpu1  :  1.8 us, 98.2 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu2  :  2.1 us, 97.9 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu3  :  1.0 us, 99.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
KiB Mem : 16266296 total,  1819300 free,  7642004 used,  6804992 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  8086580 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12902 root      20   0 1410452  32280  17744 S  48.1  0.2 751:39.59 calico-nod
    9 root      20   0       0      0      0 R  34.8  0.0 131:14.01 [rcu_sched
 3668 techstar  20   0 4816688   1.3g  23056 S  33.9  8.5 111:17.12 /usr/share
26105 root      20   0  119604   6344   2704 R  25.8  0.0   0:02.36 runc --roo
26163 root      20   0   19368    880    636 R  25.2  0.0   0:00.95 iptables-l
26150 root      20   0   18740   3136   1684 R  21.6  0.0   0:01.18 runc init
26086 root      20   0   18744   5756   2376 R  20.3  0.0   0:03.10 runc --roo
  410 root      20   0       0      0      0 S  19.4  0.0  42:42.56 [xfsaild/d
   14 root      20   0       0      0      0 S  14.8  0.0  54:28.76 [ksoftirqd
    6 root      20   0       0      0      0 S  14.2  0.0  50:58.94 [ksoftirqd
26158 root      20   0   18740   1548    936 R  14.2  0.0   0:00.90 runc --ver
31715 nfsnobo+  20   0  129972  19856   9564 S  11.3  0.1  12:41.98 ./kube-rba
10296 root      20   0 3402116 113200  39320 S  10.3  0.7   2936:50 /usr/bin/k
   22 root      rt   0       0      0      0 S   8.7  0.0   3:18.08 [watchdog/
26162 root      20   0  139592   2792   2508 R   8.4  0.0   0:00.39 /opt/cni/b
 6843 root      20   0  965824 110244  30364 S   7.7  0.7   1544:20 /usr/bin/d
   24 root      20   0       0      0      0 S   7.4  0.0  49:03.89 [ksoftirqd
 3636 techstar  20   0    4368    364    280 S   6.8  0.0   0:12.19 /tini -- /
26159 root      20   0   18740    760    552 R   6.5  0.0   0:00.28 runc --ver
 1755 root      20   0  411108   5836   4416 S   4.8  0.0  35:39.97 /usr/libex


- us cpu高的情况
[root@k8s-worker-8 ~]# top
top - 00:43:01 up 28 days,  4:31,  3 users,  load average: 72.51, 68.20, 47.01
Tasks: 263 total,   2 running, 260 sleeping,   1 stopped,   0 zombie
%Cpu0  : 77.2 us, 15.7 sy,  0.0 ni,  2.2 id,  0.0 wa,  0.0 hi,  4.8 si,  0.0 s
```

```
%Cpu1  : 77.0 us, 15.7 sy,  0.0 ni,  2.3 id,  0.0 wa,  0.0 hi,  5.0 si,  0.0 s
%Cpu2  : 70.3 us, 20.9 sy,  0.0 ni,  2.9 id,  0.0 wa,  0.0 hi,  5.9 si,  0.0 s
%Cpu3  : 76.6 us, 12.2 sy,  0.0 ni,  5.1 id,  0.0 wa,  0.0 hi,  6.1 si,  0.0 s
KiB Mem : 16266296 total,  1996620 free,  7426512 used,  6843164 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  8302092 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20072 root      20   0 7944892 689352  15924 S 137.1  4.2   3127:04 java -Dapp
29493 root      20   0 3532496 248960  17408 S  98.3  1.5   0:06.70 java -XX:+
28697 root      20   0 3711520   1.0g  18760 S  61.6  6.7 124:41.08 java -XX:+
25885 root      20   0 3716560   1.2g  18908 S  59.3  7.6 183:12.97 java -XX:+
 6843 root      20   0  965824 109568  30364 S   7.6  0.7   1544:49 /usr/bin/d
 3668 techstar  20   0 4816688   1.3g  23056 S   6.6  8.5 111:58.56 /usr/share
10296 root      20   0 3402372 111692  39320 S   6.6  0.7   2937:43 /usr/bin/k
   18 root      rt   0       0      0      0 S   5.0  0.0   5:57.54 [migration
    6 root      20   0       0      0      0 S   2.6  0.0  51:21.52 [ksoftirqd
  410 root      20   0       0      0      0 D   2.6  0.0  43:08.23 [xfsaild/d
28310 root      20   0 7807048 565740  15924 S   2.6  3.5   1036:53 java -Dapp
29741 root      20   0 7749608 540376  15848 S   2.3  3.3 304:41.47 java -Dapp
12902 root      20   0 1410452  30368  17744 S   2.0  0.2 752:30.32 calico-nod
16712 root       0 -20       0      0      0 S   2.0  0.0   1:56.16 [kworker/2
 6381 root      20   0 7782400 491476  15928 S   1.7  3.0 441:08.96 java -Dapp
```

从 sy cpu 高的 top 数据来看，这个节点显然在不断地调度系统资源，通过 top 中的 rcu_sched/softirq 等进程就可以知道，这种情况显然是因为 Kubernetes 在这个节点上过多地安排了任务。所以，我先把 Member 服务移到另一个 worker 上，然后看到 TPS 如下：



你看，TPS 增加到 400 多了，也就是说我们的方向是对的。

那为什么我们之前修改 Order 服务的 Tomcat 线程数没有看到效果呢？这是因为压力已经到了 Member 服务上，这让 Member 服务所在的 worker 节点资源使用率增加，导致

Member 服务无法正常响应请求。因此，整个 TPS 看起来没有什么优化效果。现在，我们移走了 Member 服务，看到效果明显增加，这说明我们的方向还在正确的道路上。

我们再回来看一下整体的资源监控：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	4.09 week	7.64 GiB	4	0.32	9.94%	23.04%	29.95%	0 B/s	92.61 kB/s	285	28	592.82 kbps	403.52 kbps
k8s-master-2	k8s-master-2	6.68 week	7.64 GiB	4	0.91	22.82%	25.83%	29.95%	0 B/s	96.10 kB/s	315	22	978.47 kbps	978.47 kbps
k8s-master-3	k8s-master-3	6.68 week	7.64 GiB	4	0.29	8.41%	24.09%	29.95%	0 B/s	94.65 kB/s	294	93	277.21 kbps	410.67 kbps
k8s-worker-1	k8s-worker-1	1.09 week	15.51 GiB	6	3.52	36.27%	51.01%	69.77%	0 B/s	7.29 MB/s	29	76	26.97 Mbps	27.10 Mbps
k8s-worker-2	k8s-worker-2	2.62 day	15.51 GiB	6	2.59	34.65%	18.87%	31.87%	23.02 kB/s	129.71 kB/s	28	1 K	30.68 Mbps	30.61 Mbps
k8s-worker-3	k8s-worker-3	1.17 day	15.51 GiB	6	6.89	56.08%	22.04%	50.14%	0 B/s	98.46 kB/s	28	540	38.44 Mbps	38.51 Mbps
k8s-worker-4	k8s-worker-4	5.59 week	15.51 GiB	6	8.87	62.34%	37.76%	48.04%	9.21 kB/s	148.32 kB/s	51	21 K	15.31 Mbps	20.10 Mbps
k8s-worker-5	k8s-worker-5	5.59 week	15.51 GiB	6	4.78	45.06%	46.65%	37.64%	18.41 kB/s	10.20 kB/s	28	150	18.07 Mbps	27.07 Mbps
k8s-worker-6	k8s-worker-6	5.58 week	15.51 GiB	6	0.54	13.88%	21.30%	39.76%	0 B/s	1.58 kB/s	29	121	916.36 kbps	870.20 kbps
k8s-worker-7	k8s-worker-7	5.29 week	15.51 GiB	4	4.06	56.59%	29.23%	42.91%	13.81 kB/s	135.62 kB/s	29	677	23.10 Mbps	22.99 Mbps
k8s-worker-8	k8s-worker-8	4.11 week	15.51 GiB	4	0.37	12.19%	29.04%	46.58%	0 B/s	7.18 kB/s	27	202	18.19 Mbps	18.28 Mbps
k8s-worker-9	k8s-worker-9	5.29 week	15.51 GiB	6	2.64	26.84%	27.09%	29.95%	4.60 kB/s	21.70 kB/s	28	76	47.49 Mbps	57.74 Mbps

现在，没有一个 worker 的资源用满或者接近用满，完全不符合我们“把资源都用起来”的目标，这显然是不可接受的。

在长时间的压力中，我们发现资源怎么也用不上去。而且在上节课第三阶段的最后一个图中，你也能清楚地看到这一点。

到这里为止，我们查看了一次次的性能分析决策树，也调整了一些参数，比如 Spring Boot 中的 Tomcat 连接池、JDBC 池、Jedis 池、MQ 池等，调整之后 TPS 似乎有增加的趋势，但是非常不明显。所以，我们只能开始新一轮的定位。

第五阶段

定位时间消耗

在上一个阶段的分析中，我们用日志拆分了响应时间，是想让你看到我们用不同的手段都可以把响应时间拆出来。这也是我一直强调的：你不要在意用什么手段，而要在意你想要的是什么。

在这一阶段中，我们再换一个思路：跟踪方法的执行过程来判断时间消耗。我想让你看到：在优化过程中，唯有思路不变，手段任你选择。

这个方法和我们用日志拆分时间的逻辑其实是一样的。我们可以直接用 Arthas 来定位方法的时间消耗。请你记住，除了 Arthas 之外，还有很多其他工具也是可以的，比如 JvisualVM/JMC/BTrace 等。

我们已经知道接口方法是 com.dunshan.mall.order.service.impl.PortalOrderServiceImpl 中的 generateOrder，所以，我们直接 trace（跟踪）它就可以了。

你要注意，在这一步中，我们需要反复 trace 多次，这是为了保证判断方向的正确性。不得不承认，这是一种耗时又枯燥的工作，有一种数羊睡觉的感觉。不过，有的人能数睡着，有的人却是越数越兴奋。

现在，我们来看一下反复 trace 后的结果。由于跟踪的栈实在太长了，我把多次跟踪的内容做了简化，其中重要的部分如下所示：

```
+---[91.314104ms] com.dunshan.mall.order.feign.MemberService:getCurrentMember(
.....................
+---[189.777528ms] com.dunshan.mall.order.feign.CartItemService:listPromotionn
.....................
+---[47.300765ms] com.dunshan.mall.order.service.impl.PortalOrderServiceImpl:s
```

为什么说这几个方法重要呢？这里我要说明一下，对于跟踪的内容，我们主要判断的是：**消耗时间的方法是不是固定的**。如果时间不是消耗在了固定的方法上，那就有些麻烦了，因为这说明不是方法本身的问题，而是其他的资源影响了方法的执行时间；如果时间一直消耗在了固定的方法上，就比较容易了，我们只要接着去跟踪这个方法就好了。

而我反复跟踪了多次之后，总是发现上面几个方法都比较消耗时间。既然已经知道了方法的时间消耗，那全局监控已经救不了我们了，只有在定向监控中来分析了。

定向监控分析

我先说明一下，根据我们的分析思路，我在定向监控分析之前，反复分析了全局监控计数器，没觉得有什么资源使用上的问题。并且从压力工具到数据库，我也没发现有什么阻塞点，整条大路都很宽敞。

但是，上面我们也确实看到了响应时间消耗在了几个具体的方法上，并且这几个方法并不是一直都消耗这么长的时间，而是有快有慢。

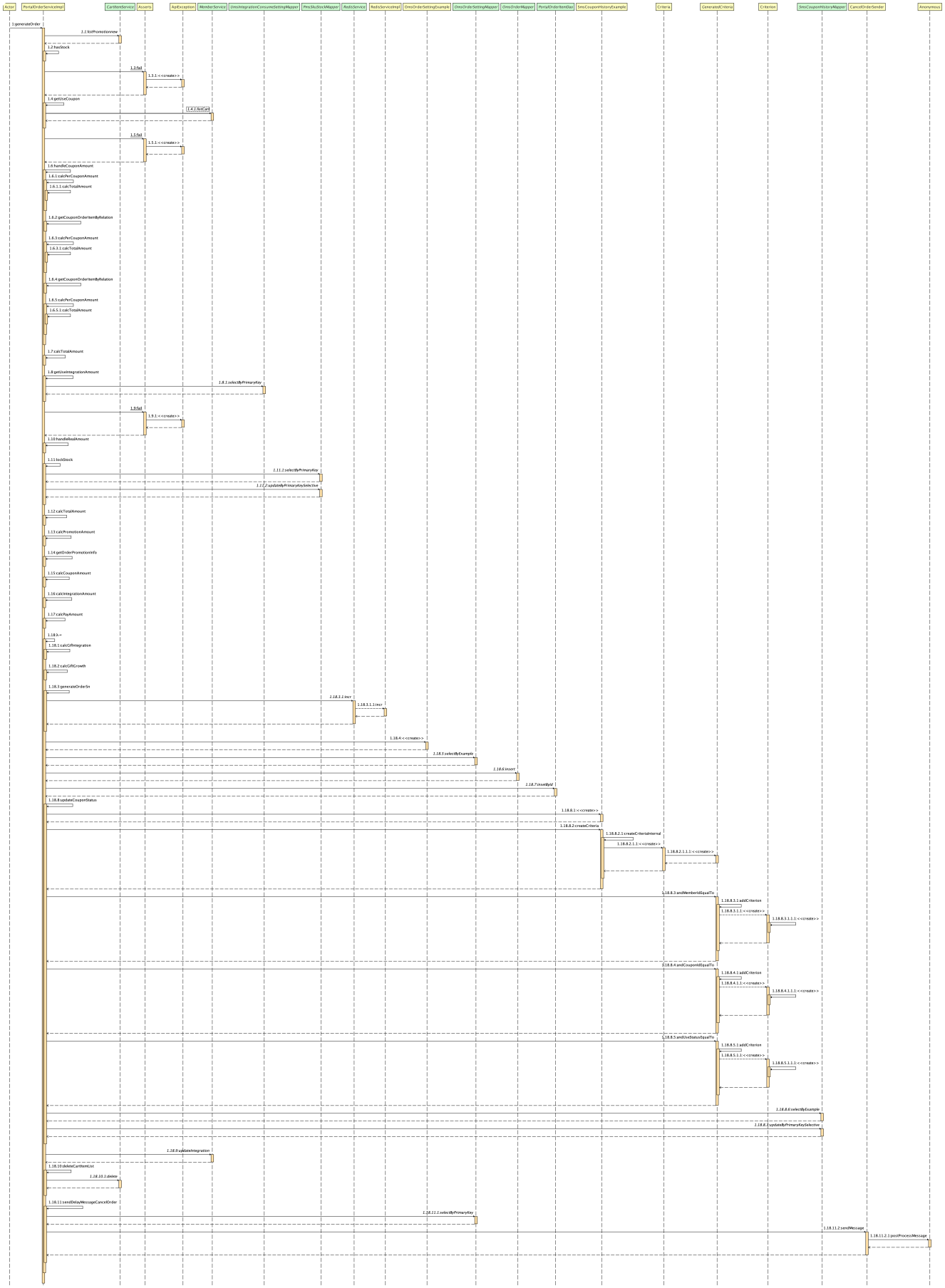
经过反复确认后，我觉得有必要来看一下业务逻辑了。因为对于一个复杂的业务来说，如果业务代码逻辑太长，那我们不管怎么优化，都不会有什么效果，最后只能按照扩容的思路来加机器了。

不过，在我的逻辑中，即便是加机器，我们也要给出加机器的逻辑。如果业务可优化，那我们更要尽力一试。因为从成本上来说，优化代码是一个更优的选择。

在这里，我多说几句闲话。我看到过很多企业连一些简单的优化都没有做，就从寻找心理安全感的角度去增加机器，结果耗费了大量的成本，这是非常不理智的。从技术的角度来说，花不多的时间成本就可以节省大量的资源成本，这显然是很划算的。可是，受一些社会不良思维的误导，有些企业就觉得只要能通过加机器解决的问题，都不是啥大问题。

对于这种思路，我们就得放到成本上来算一算了。大部分打工人可能会觉得，反正用的又不是自己的钱，管老板花多少钱加机器干嘛？没意义。但是，从节能减排的全球大局观来看，一个该做的优化没有做，不仅浪费公司的成本，还一点儿都不环保！呃..... 好像扯的有点远了。

我们回到正题，既然我们想优化业务，就得先知道业务的调用逻辑是个啥样子。所以我们打开 idea，找到 generateOrder 方法，然后把 sequence diagram（idea 的一个插件）打开，就看到了这样一张很长的业务逻辑图：



如果你看不懂这张图，也没有关系。我在这里给你大致描述一下这张图里有什么东西：

1. 获取用户名；

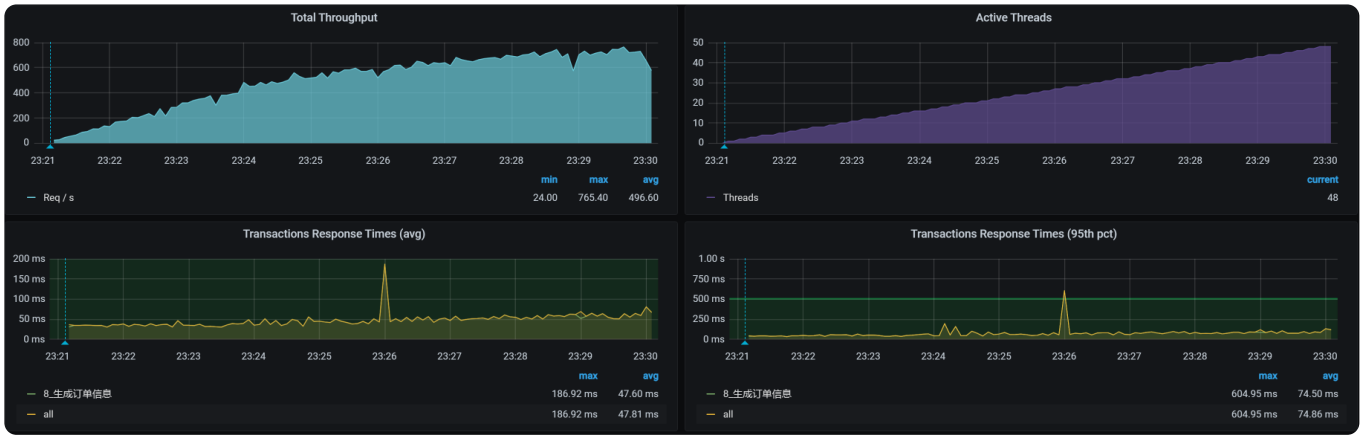
2. 获取购物车列表；
3. 获取促销活动信息；
4. 判断库存；
5. 判断优惠卷；
6. 判断积分；
7. 计算金额；
8. 转订单并插库；
9. 获取地址信息；
10. 计算赠送积分和成长值
11. 插入订单表；
12. 更新优惠卷状态；
13. 扣积分；
14. 删除购物车商品；
15. 发送取消订单消息；
16. 返回结果；

是不是有种很复杂的感觉？通过我大概列出来的这些内容，你就能知道下订单这个动作有多长了。对这样的复杂接口，如果业务逻辑要求必须是这样的，那我们在接口上就没有什么优化空间了。在前面，我们已经把 TPS 优化到了 400 多，在这样的硬件机器上，也基本上就这样了。

在这节课中，我们不是要去设计一个下订单的业务逻辑，因为每个企业的下订单逻辑，都会有不同的业务限制。做为性能工程师，我们没有对业务逻辑的设计说改就改的权利，因为修改业务逻辑需要所有的相关人员一起商讨确定。不过，我们可以通过分析的结果给出优化的建议。

在这里，我把优惠卷、积分、发送延时取消订单信息的步骤都从下订单的步骤中删掉。有人可能会问这样改合适吗？我强调一下，不是我要这样改业务逻辑，而是想看看这样改了之后，TPS 有没有增加。如果增加了，就说明我们的方向是对的，也就是说，这个业务逻辑需要再和各方商量一下，重新设计。

我们来看修改之后的 TPS 图：



可以看到，这样的修改确实有效果，那我们后续的优化建议就比较清晰了。如果你在项目中遇到这样的接口，优化建议就可以这样来提：

1. 分库分表；
2. 利用缓存；
3. 异步处理非关键步骤；
4. 大接口拆成小接口。

但是，建议终归是建议，通常在一个企业中，对于这样的接口，技术团队会根据具体的业务逻辑做长时间的技术分析，来判断如何实现。如果确实没办法在技术上做优化，那就只能上最后一招：扩容！这个扩容就不再是扩某一段了，而是一整条链路上涉及到的服务。

还有一点，在一个业务链路中，每个企业通常都是根据发展的速度做相应的技术沉淀。如果技术团队太追潮流了，学习成本大，不见得是好事；如果太陈旧了，维护的成本大，也不见得是好事。因此，我们只有根据实际的业务发展不断地演进业务流程和技术实现，才是正道。

我们优化到这里，看似就可以结束收工了，但是并没有，因为天不随人愿的定律从来都没有被打破过，我们只得来到第六个阶段。

第六阶段

定位 TPS 会降下来的问题

具体是什么原因呢？我在接着压的时候，又出现了这样的问题：

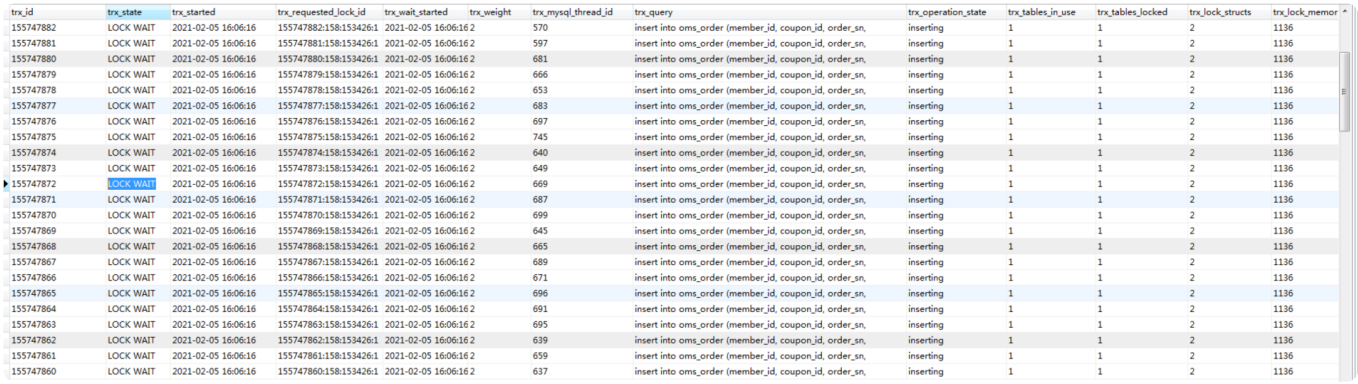
你看，TPS 掉呀掉的，心都碎了……虽说在每个项目的优化过程中，都会出现各种意外的事情，但是，这个接口的意外也太多了点。没办法，我们接着查吧。

仍然是按照高老师强调的性能分析决策树（如果你不清楚，可以再看看第 4 讲），我们一个个计数器看过去，最后在 mysqlreport 中看到了下面这些数据：

复制代码

```
__ InnoDB Lock _____________________________________________________
Waits              889      0.1/s
Current             77
Time acquiring
  Total       36683515 ms
  Average        41263 ms
  Max            51977 ms
```

显然当前的锁有点多，并且这锁的时间还挺长。要想查锁，就得先知道当前正在运行的是什么样的事务，所以我们就去查一下 innodb_trx 表，因为 MySQL 在这个表中会记录所有正在执行的事务。在数据库中，我们发现了大量的 lock_wait（锁等待）：



trx_id	trx_state	trx_started	trx_requested_lock_id	trx_wait_started	trx_weight	trx_mysql_thread_id	trx_query	trx_operation_state	trx_tables_in_use	trx_tables_locked	trx_lock_structs	trx_lock_memor
155747882	LOCK WAIT	2021-02-05 16:06:16	155747882:158:153426:1	2021-02-05 16:06:16	2	570	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747881	LOCK WAIT	2021-02-05 16:06:16	155747881:158:153426:1	2021-02-05 16:06:16	2	597	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747880	LOCK WAIT	2021-02-05 16:06:16	155747880:158:153426:1	2021-02-05 16:06:16	2	681	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747879	LOCK WAIT	2021-02-05 16:06:16	155747879:158:153426:1	2021-02-05 16:06:16	2	666	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747878	LOCK WAIT	2021-02-05 16:06:16	155747878:158:153426:1	2021-02-05 16:06:16	2	653	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747877	LOCK WAIT	2021-02-05 16:06:16	155747877:158:153426:1	2021-02-05 16:06:16	2	683	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747876	LOCK WAIT	2021-02-05 16:06:16	155747876:158:153426:1	2021-02-05 16:06:16	2	697	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747875	LOCK WAIT	2021-02-05 16:06:16	155747875:158:153426:1	2021-02-05 16:06:16	2	745	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747874	LOCK WAIT	2021-02-05 16:06:16	155747874:158:153426:1	2021-02-05 16:06:16	2	640	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747873	LOCK WAIT	2021-02-05 16:06:16	155747873:158:153426:1	2021-02-05 16:06:16	2	649	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747872	LOCK WAIT	2021-02-05 16:06:16	155747872:158:153426:1	2021-02-05 16:06:16	2	669	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747871	LOCK WAIT	2021-02-05 16:06:16	155747871:158:153426:1	2021-02-05 16:06:16	2	687	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747870	LOCK WAIT	2021-02-05 16:06:16	155747870:158:153426:1	2021-02-05 16:06:16	2	699	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747869	LOCK WAIT	2021-02-05 16:06:16	155747869:158:153426:1	2021-02-05 16:06:16	2	645	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747868	LOCK WAIT	2021-02-05 16:06:16	155747868:158:153426:1	2021-02-05 16:06:16	2	665	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747867	LOCK WAIT	2021-02-05 16:06:16	155747867:158:153426:1	2021-02-05 16:06:16	2	689	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747866	LOCK WAIT	2021-02-05 16:06:16	155747866:158:153426:1	2021-02-05 16:06:16	2	671	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747865	LOCK WAIT	2021-02-05 16:06:16	155747865:158:153426:1	2021-02-05 16:06:16	2	696	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747864	LOCK WAIT	2021-02-05 16:06:16	155747864:158:153426:1	2021-02-05 16:06:16	2	691	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747863	LOCK WAIT	2021-02-05 16:06:16	155747863:158:153426:1	2021-02-05 16:06:16	2	695	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747862	LOCK WAIT	2021-02-05 16:06:16	155747862:158:153426:1	2021-02-05 16:06:16	2	639	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747861	LOCK WAIT	2021-02-05 16:06:16	155747861:158:153426:1	2021-02-05 16:06:16	2	659	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
155747860	LOCK WAIT	2021-02-05 16:06:16	155747860:158:153426:1	2021-02-05 16:06:16	2	637	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136

既然有锁等待，那我们自然要查一下锁关系，看看是什么在等待锁。在看具体的锁关系之前，我们也来查看一下应用日志。

为什么要看应用日志呢？因为对于数据库来说，锁是为了保护数据的一致性，而产生锁的事务自然是从应用中来的。按照这样的逻辑，我们在 MySQL 中和在应用中看到的事务 SQL，应该是对应的。而现在我们只看到了 MySQL 中的锁，还不知道在应用中会是什么样子，所以，我们要看一下应用的日志。

这里温馨提醒一句：**在这个时候，我们还需要注意，不要用重压力工具中的某些具有唯一性的参数化数据。因为当参数化数据用重了，在数据库中执行 update 语句也照样会出现锁。**

在查看了应用日志之后，我们看到如下信息：

```
[2021-02-06 00:46:59.059] [org.apache.juli.logging.DirectJDKLog] [http-nio-808
### Error updating database.  Cause: com.mysql.cj.jdbc.exceptions.MySQLTransac
### The error may involve com.dunshan.mall.mapper.OmsOrderMapper.insert-Inline
### The error occurred while setting parameters
### SQL: insert into oms_order (member_id, coupon_id, order_sn,        create_
### Cause: com.mysql.cj.jdbc.exceptions.MySQLTransactionRollbackException: Loc
; Lock wait timeout exceeded; try restarting transaction; nested exception is
com.mysql.cj.jdbc.exceptions.MySQLTransactionRollbackException: Lock wait time
```

你看，连一个 insert 都会报 lock_wait，这显然是出现表级锁了。因为 insert 本身是不会出现表级锁的，所以应该还有其他的信息。我们接着看日志，果然，又看到如下信息：

```
[2021-02-06 01:00:51.051] [org.springframework.scheduling.support.TaskUtils$Lo
```

```
org.springframework.dao.CannotAcquireLockException:
### Error updating database.  Cause: com.mysql.cj.jdbc.exceptions.MySQLTransac
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: update oms_order         set status=?         where id in          (
### Cause: com.mysql.cj.jdbc.exceptions.MySQLTransactionRollbackException: Loc
; Lock wait timeout exceeded; try restarting transaction; nested exception is
```

你看，其中有 update 语句，这样一来，逻辑就成立了：我们知道，update 是会锁数据的，但是，MySQL 用的是 InnoDB 的引擎。如果 update 的条件是精确查找，那就应该不会出现表级锁。

可是，如果 update 的范围比较大，就会有问题了，因为这会导致 insert 语句被阻塞。过一会儿之后，你就会看到如下内容：

```
SELECT * from information_schema.INNODB_TRX;
select * from information_schema.innodb_locks;
select * from information_schema.INNODB_LOCK_WAITS;
```

信息 | 结果1 | 结果2 | 结果3 | 概况 | 状态

trx_id	trx_state	trx_started	trx_requested_lock_id	trx_wait_started	trx_weight	trx_mysql_thread_id	trx_query	trx_operation_state	trx_tables_in_use	trx_tables_locked	trx_lock_structs	trx_lock_me
158028668	LOCK WAIT	2021-02-05 17:53:22	158028668:158:380287:1	2021-02-05 17:53:22	2	702	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028667	LOCK WAIT	2021-02-05 17:53:22	158028667:158:380287:1	2021-02-05 17:53:22	2	575	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028666	LOCK WAIT	2021-02-05 17:53:22	158028666:158:380287:1	2021-02-05 17:53:22	2	680	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028665	LOCK WAIT	2021-02-05 17:53:22	158028665:158:380287:1	2021-02-05 17:53:22	2	663	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028664	LOCK WAIT	2021-02-05 17:53:22	158028664:158:380287:1	2021-02-05 17:53:22	2	766	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028663	LOCK WAIT	2021-02-05 17:53:22	158028663:158:380287:1	2021-02-05 17:53:22	2	637	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028662	LOCK WAIT	2021-02-05 17:53:22	158028662:158:380287:1	2021-02-05 17:53:22	2	558	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028661	LOCK WAIT	2021-02-05 17:53:22	158028661:158:380287:1	2021-02-05 17:53:22	2	581	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028660	LOCK WAIT	2021-02-05 17:53:22	158028660:158:380287:1	2021-02-05 17:53:22	2	664	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028659	LOCK WAIT	2021-02-05 17:53:22	158028659:158:380287:1	2021-02-05 17:53:22	2	638	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028658	LOCK WAIT	2021-02-05 17:53:22	158028658:158:380287:1	2021-02-05 17:53:22	2	658	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028657	LOCK WAIT	2021-02-05 17:53:22	158028657:158:380287:1	2021-02-05 17:53:22	2	873	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028656	LOCK WAIT	2021-02-05 17:53:22	158028656:158:380287:1	2021-02-05 17:53:22	2	698	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028655	LOCK WAIT	2021-02-05 17:53:22	158028655:158:380287:1	2021-02-05 17:53:22	2	598	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028654	LOCK WAIT	2021-02-05 17:53:22	158028654:158:380287:1	2021-02-05 17:53:22	2	692	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028653	LOCK WAIT	2021-02-05 17:53:22	158028653:158:380287:1	2021-02-05 17:53:22	2	571	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028652	LOCK WAIT	2021-02-05 17:53:22	158028652:158:380287:1	2021-02-05 17:53:22	2	784	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028651	LOCK WAIT	2021-02-05 17:53:22	158028651:158:380287:1	2021-02-05 17:53:22	2	597	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028650	LOCK WAIT	2021-02-05 17:53:22	158028650:158:380287:1	2021-02-05 17:53:22	2	874	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028649	LOCK WAIT	2021-02-05 17:53:22	158028649:158:380287:1	2021-02-05 17:53:22	2	683	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028648	LOCK WAIT	2021-02-05 17:53:22	158028648:158:380287:1	2021-02-05 17:53:22	2	701	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028647	LOCK WAIT	2021-02-05 17:53:22	158028647:158:380287:1	2021-02-05 17:53:22	2	703	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028646	LOCK WAIT	2021-02-05 17:53:22	158028646:158:380287:1	2021-02-05 17:53:22	2	662	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028645	LOCK WAIT	2021-02-05 17:53:22	158028645:158:380287:1	2021-02-05 17:53:22	2	591	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136
158028644	LOCK WAIT	2021-02-05 17:53:22	158028644:158:380287:1	2021-02-05 17:53:22	2	559	insert into oms_order (member_id, coupon_id, order_sn,	inserting	1	1	2	1136

我们看到，所有的 insert 都在 LOCK WAIT 状态了，这就是表级锁对 insert 产生的影响。如果你再查一下锁和锁等待的话，就会看到如下信息：

```
17 SELECT * from information_schema.INNODB_TRX;
18 select * from information_schema.innodb_locks;
19 select * from information_schema.INNODB_LOCK_WAITS;
20
```

信息 | 结果1 | 结果2 | 结果3 | 概况 | 状态

lock_id	lock_trx_id	lock_mode	lock_type	lock_table	lock_index	lock_space	lock_page	lock_rec	lock_data
158028668:	158028668	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
157710723:	157710723	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028667:	158028667	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028666:	158028666	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028665:	158028665	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028664:	158028664	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028663:	158028663	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028662:	158028662	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028661:	158028661	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028660:	158028660	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028659:	158028659	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028658:	158028658	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028657:	158028657	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028656:	158028656	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028655:	158028655	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028654:	158028654	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028653:	158028653	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028652:	158028652	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028651:	158028651	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028650:	158028650	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028649:	158028649	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028648:	158028648	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028647:	158028647	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028646:	158028646	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028645:	158028645	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record
158028644:	158028644	X	RECORD	`mall`.`oms_order`	GEN_CLUST_INDEX	158	380287	1	supremum pseudo-record

不难看出，lock_mode 这一列的值全是 X，意思是 X 锁。我们知道，排他锁（X 锁），又叫写锁。图中的锁类型（lock_type）全是 RECORD，锁住的是索引，并且索引是 GEN_CLUST_INDEX，说明这个锁等待是因为 innodb 创建的隐藏的聚集索引。

当一个 SQL 没有走任何索引时，就会在每一条聚集索引后面加 X 锁，这和表级锁的现象是一样的，只是在原理上有所不同而已。为了方便描述，我们仍然用“表级锁”来描述。

要查锁，就得看看是谁持有锁。经过仔细查看上面的 INNODB_LOCK_WAIT 后，我们确定了这个锁对应的事务 ID 是 157710723，它对应的 SQL 是：

复制代码

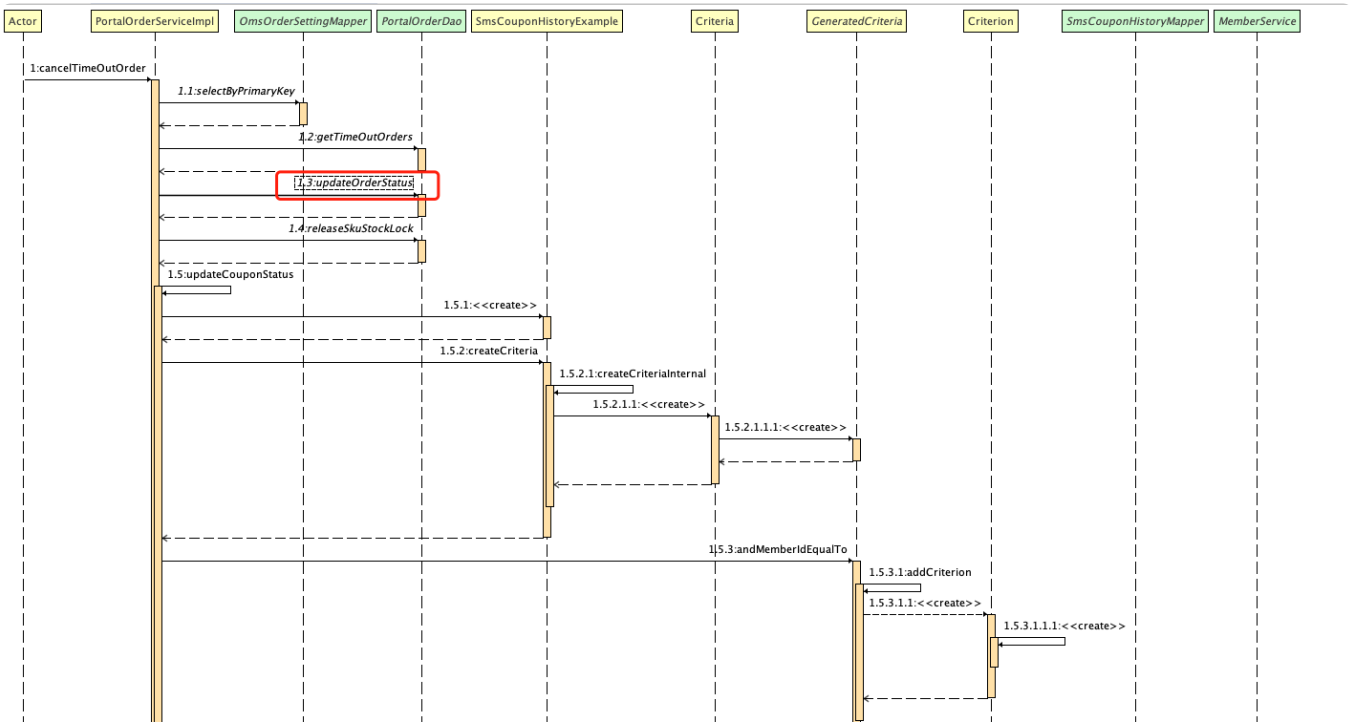
```
1 update oms_order set status=4 where id in  (        0  );
```

我们去代码中查一下这段 update 代码：

复制代码

```
1 /**
2  * 批量修改订单状态
3  */
4 int updateOrderStatus(@Param("ids") List<Long> ids,@Param("status") Integer st
```

原来这是一个批量任务的调用，具体逻辑如下：



这个批量任务的问题在于，在一个订单表中做批量更新的操作，并且这个批量查询的内容还挺多。因为上面的 ID 是 0，表示订单是未支付的，而未支付的订单在这个表中有很多，所以，在更新时做大范围的查找，会进而导致表级锁的出现。

这个批量任务的设计明显有问题。你想想，要在订单表中做更新这么大的动作，那也应该是做精准更新，而不是范围更新。其实对于订单的更新逻辑，我们可以选择其他的实现方式。

锁的原因找到了，我们现在要把范围更新改为非常精准的更新，让它不产生表级锁。修改之后，重新执行场景的结果如下：

从优化效果来看，TPS 已经达到 700 以上了。对这样一个复杂的接口来说，这已经非常不错了。

其实，这个订单业务还有很多的优化空间，比如说：

1. 异步生成订单序列号，然后存放到 Redis 里，随用随取。
2. 批量业务需要重新设计。
3. 读写分离之后，对业务代码也做相应更新。
4. ……

由于订单逻辑是电商中的非常复杂的一步，我就不再展开说了，因为再说就超出了性能的范畴。

总结

在这个接口中，我们遇到了好几个问题。先抛开问题和复杂度不说，我想表达的是，在性能优化过程中，问题是像洋葱一样一个个剥开的。虽然有可能一个优化动作就可以产生很好的效果，但是我们一定不要着急，要慢慢分析一个个问题。

回顾一下我们对这个接口的所有分析优化过程。在第一阶段中，我们修改线程池产生了效果，但也出现了新问题；在第二阶段中，我们解决了查询大量数据导致内存被耗光的问题；在第三介阶段，我们解决了索引的问题；在第四阶段中，我们重新调配了资源，让系统的调度更加合理。

在第五阶段中，我们定位了方法的时间消耗问题，这一步你要注意，一定要在分析了业务逻辑之后再做相应的优化，不要因一味追求性能的优化效果而纠结。

在第六阶段中，我们定位了批量任务设计不合理的问题。在正经的批量任务中，批量产生的表级锁和 insert 的功能点，一定要分开。

总之，在分析的过程中，我们不要界定问题的边界，遇到什么问题就解决什么问题，不急不燥，不卑不亢。

课后作业

最后，请你思考两个问题：

1. 如何快速定位内存被消耗光的情况？

2. 如何快速定位业务逻辑导致的 TPS 上不去、资源也用不上的情况？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 生成订单信息之一：应用JDBC池优化和内存溢出分析

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。